An Asynchronous Architecture for Modeling Intersegmental Neural Communication

Girish N. Patel, *Member, IEEE*, Michael S. Reid, *Member, IEEE*, David E. Schimmel, *Senior Member, IEEE*, and Stephen P. DeWeerth, *Senior Member, IEEE*

***Abstract*—This paper presents an asynchronous VLSI architecture for modeling the oscillatory patterns seen in segmented biological systems. The architecture emulates the intersegmental synaptic connectivity observed in these biological systems. The communications network uses address-event representation (AER), a common neuromorphic protocol for data transmission. The asynchronous circuits are synthesized using communicating hardware processes (CHP) procedures. The architecture is scalable, supports multichip communication, and operates independent of the type of silicon neuron (spiking or burst envelopes). A 16-segment prototype system was developed, tested, and implemented; data from this system are presented.**

***Index Terms*—Address event representation (AER), asynchronous circuits, central pattern generator (CPG), neurobiological modeling, neuromorphic engineering, silicon neuron, VLSI architecture.**

I. Introduction

BIOLOGICAL systems perform robust neural computations largely because of their highly complex communication networks. For example, the neural systems that generate and modulate axial locomotion in segmented invertebrates are facilitated by short- and long-distance synaptic connections along the length of the animal [1]. We develop neuromorphic implementations [2] of these segmented systems with a focus on communication architectures that model the intersegmental connections.

The design of an architecture that is capable of producing the variety of oscillatory patterns seen in segmented biological systems must draw inspiration from the observed and hypothesized properties present in biology. An architecture that incorporates these properties can be used to validate the principles underlying intersegmental coordination and can result in the development of engineered systems that reproduce the complex behaviors of their biological counterparts. The properties in question fall into two categories: 1) intersegmental connectivity and 2) intrasegmental neuronal properties and synaptic interactions.

Intersegmental connectivity in segmented systems creates coordination among the neural oscillators in the spinal cord. These oscillators, often called central pattern generators (CPG), are present at each segment along the animal's body; CPGs generate rhythmic, oscillatory patterns to produce complex activation of different groups of motor neurons. The pattern-generating circuits are densely connected. For example, in the lamprey swim system, axonal projections can extend up to nearly half the length of the body (up to 50 segments), implying that a typical segment may receive hundreds of connections [1].

A realistic model of the biological system should include the ability to create different CPG configurations and to implement both short- and long-distance neural connections. These synaptic interconnections are essential for the coordination of segmental oscillators resulting in the movement of the animal. The intersegmental connectivity and parameter space can be greatly simplified by assuming that the neural signals exhibit uniform delay and translational invariance of synaptic connections. This translational invariance, referred to as *synaptic spread* in the biological literature, implies that for each connection a neuron makes with other neurons in its own segment, the neuron makes the same connections with homolog neurons in neighboring segments [3]. This simple connectivity rule exists in intersegmental biological systems.

An unnecessarily large amount of silicon real estate would be necessary to emulate the same level of connectivity with individual wires in a very large scale integration (VLSI) system. In addition, the conduction velocity of signals in metallic wires is too fast to mirror the delays present in the biological systems. Clearly, wires in biology and VLSI systems are grossly mismatched, whereas wires in biology are slow and densely packed, their VLSI counterparts are fast and occupy valuable silicon real estate [2].

In order to resolve this mismatch between the wiring in biological and VLSI systems, neuromorphic engineers often use address-event representation (AER) [4], [5], a mechanism that has been utilized in neuromorphic analog VLSI systems. In this protocol, action potentials (or other events such as burst envelopes) are encoded and time-multiplexed over a high-speed communications channel. Because the timing of events in biological systems are unquantized, AER architectures typically transmit data asynchronously; thus, information is represented by the origin and timing of individual events. This information is preserved as long as the bandwidth of the communications system is much greater than the rate at which events are generated.

In this paper, we describe a custom communications architecture that we have developed for specific use in hardware models of intersegmental coordination [6]. To match its biological counterpart, including the constraints due to intersegmental coordination, we implemented a unique AER architecture that uses a pipelined broadcast scheme to emulate a large number of intersegmental connections with distance-dependent delays. The architecture is scalable, supports multichip com-

Manuscript received August 30, 2004; revised May 18, 2005. This work was supported by the National Science Foundation (NSF) under Grant IBN-9511721 and Grant IBN-0131612.

The authors are with the Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: steve.deweerth@ece.gatech.edu).

Digital Object Identifier 10.1109/TVLSI.2005.863762

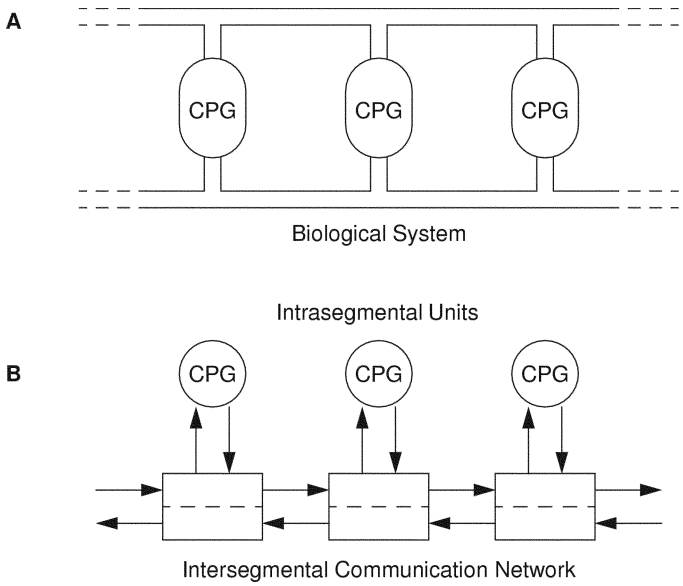

Fig. 1. Conceptual views of the (A) biological and (B) artificial segmental architectures. The intersegmental communications network of the artificial system facilitates communication among the intrasegmental units with pipelined stages.

munication, and operates independently of the type of silicon neuron (spiking or burst envelopes).

II. System Architecture

Our architecture and the biological system on which it is based are shown in Fig. 1. The system is divided into intrasegmental units and an intersegmental connection network. The intrasegmental units consist of a number of interconnected bursting neurons, each of which generates action potentials (or burst envelopes) and has the potential to receive both intrasegmental and intersegmental synaptic input. The intersegmental connection network converts events from the intrasegmental units (action potentials or burst envelopes) into packets of data that are stored and then transmitted to other intrasegmental units, where the information contained in the packets has an effect on the intersegmental activity.

Researchers have developed address-event systems that use either partitioned or global bus architectures that broadcast each event to a large number (or all) of the other neurons [4], [5]. These architectures are not efficient for our application because our system requires that each event be capable of synapsing on every other segment (for generality) with a delay that is linear in distance. This delay requirement would make the decoding in a broadcast bus scheme very difficult. Therefore, to fulfill our requirements, we developed a pipelined broadcast scheme that is a direct parallel to its biological counterpart. The principal novelty of our address-event scheme is that both addresses and delays are generated implicitly from the system architecture.

In the architecture, each event is passed from segment to neighboring segment bidirectionally down the length of the one-dimensional communications network. By delaying each event at every segment, the pipeline architecture facilitates the creation of distance-dependent delays. These delays model the delays present in the neural fibers that exist along the length of the animal's body.

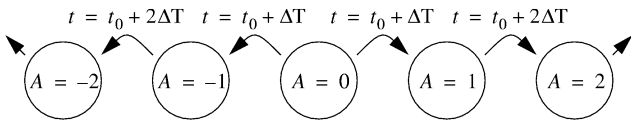

Fig. 2. Relative addressing scheme showing the origination of distance-dependent delays.

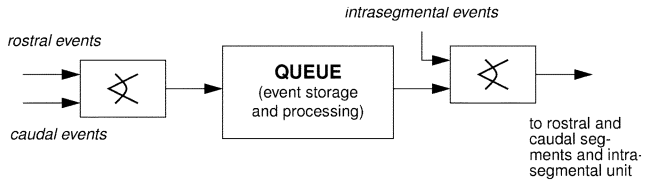

Fig. 3. Block-level diagram of a communications node illustrating how events enter and exit each stage of the pipeline.

The other primary advantage of this architecture is that it can easily generate a relative addressing scheme (as opposed to an absolute addressing scheme that would be required in a global bus architecture). By using relative addressing in our architecture, we are able to implement synaptic spread—translational invariance of synaptic connections.

Fig. 2 illustrates the event-passing architecture with respect to the relative addressing and distance-dependent delays. Each event, generated at a particular node (the center node, in this example), is transmitted bidirectionally down the length of the network. It is delayed by time ΔT at each segment, not including the initiating segment. By using a sorted queue, multiple delays can be achieved; thus, axons with different conduction velocities can be implemented. If all conduction velocities are equal, the sorted queue can be replaced by the simpler first-in-first-out (FIFO) queue. In either queue, as an event arrives at each new segment, it is time stamped, its relative address is incremented (or decremented), and then it is stored in a queue for the interval ΔT. As the event exits the queue, its data is decoded by the intrasegmental units, and synaptic inputs are applied to the appropriate intrasegmental neurons. To evoke a correct type of response, the address of each event also contains information about its *event type*, which can represent information such as which neuron fired and whether the event should propagate in the ascending or descending direction.

A. Routing Events

A block-level diagram of a single communications node illustrating how events enter and exit each stage of the pipeline is shown in Fig. 3. Because events arriving from neighboring segments are merged and inserted into the queue of a local segment, and events that exit the queue are sent back to the neighboring segments, it is possible for events to circulate around segments indefinitely. To prevent this possibility, it is necessary to detect circling events, and when detected, to prevent them from entering the queue. This is accomplished by tagging events with a direction bit (a/d) that describes whether the event is ascending (propagating toward the head, $a/d = 1$) or whether it is descending (propagating toward the tail, $a/d = 0$). An event is dropped, for example on the descending port (a port designated for receiving descending events from the rostral segment), if the event is of type ascending.

The benefit of a single queue instead of two queues (for events propagating in opposite directions) is manifested when the utilization of the queue is considered at each segment. If two queues are used, the queues at the boundaries of the system would either be overutilized or underutilized, depending on the direction and the position of the queue. For example, in the head segment of the system, the queue storing the descending events would remain unoccupied, while the queue storing the ascending events would be heavily occupied. Thus, by storing both ascending and descending events in a single queue, the occupancy of the queues along the length of the system will, on average, be uniform.

Another advantage of using a single queue is that the number of I/O signals between each pipeline stage is reduced. The use of two queues would require two input ports and two output ports, whereas in the single-queue design, two input ports and a single output port are required. In our design, the events that exit the stage are received by three input ports: the input port of the local intrasegmental unit and the input ports of the two neighboring segments.

Although local synaptic connections can be hard-wired in the intrasegmental units, generally, it should also be possible to establish local CPG connections via the communications network. Since local connections are fast, the CPG events are inserted at the tail of the queue, bypassing the intersegmental delay. These events are immediately received by the local intrasegmental units in which appropriate action can be taken. An advantage of this feature is that the circuit implementation for both local and long-distance weights is the same.

B. Event Types

One of our goals is to use different intrasegmental units without redesigning the network. This constraint implies a generic interface between the intrasegmental units and the intersegmental communications network. One method we use to implement a generic interface is to append additional bits of data to each event address. The additional bits encode an event type that carries information about which intrasegmental neuron generated the event and what kind of response should be evoked. Because the event types are meaningful only to the intrasegmental units, they are encoded by the intrasegmental units and are transmitted from segment to segment without interpretation by the intersegmental communications network.

We use Morris–Lecar neurons, implemented in silicon and whose outputs are burst envelopes, to construct the half-center oscillators that form the intrasegmental units. In order to evoke an appropriate synaptic response, we encode events as rising or falling based on the change in a neuronal membrane potential. This information is encoded by feeding both inverted and noninverted versions of the neuronal membrane potential into the input of an encoder. The encoding of event types includes information describing which neuron fired $(NNum)$, whether the event was a rising-edge or a falling-edge event (r/f), and whether the event should propagate in the ascending or descending direction (a/d). Because we want to establish connections in both directions, each transition in a neuron's membrane potential generates both ascending and descending event types. The relative address of all events generated by local intrasegmental units is zero.

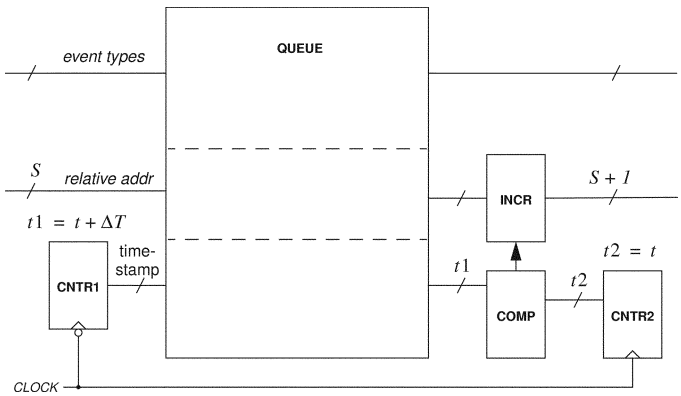

Fig. 4. Processing of event data at each node of the communications network (handshaking signals were omitted for clarity).

C. Processing Events

In each stage of the pipeline, events must be stored in a queue for the ΔT interval, and before they exit the stage, their relative addresses must be incremented. The section of the stage that processes the events is shown in Fig. 4. This section contains two counters, a queue, a comparator, and an incrementer. The two counters run at the same clock speed; however, because they are initialized to different values, their outputs display an offset: $t1 - t2 = \Delta T$. This offset corresponds to the intersegmental delay.

An event is processed as follows. When an event arrives from a neighboring segment, it is time stamped with the contents of the first counter, $t1 = t + \Delta T$, and subsequently, inserted into the head of the queue. At the tail of the queue, the event's time stamp is compared with the contents of the second counter, $t2 = t$. When $t1 = t2$, the event is allowed to exit the queue. Because we are emulating axons with uniform conduction velocities, events exit the queue in the same order as they enter. However, before exiting the stage, the event's relative address is incremented by the incrementer. We implement the delaying of events with two counters instead of one counter because the implementation is simplified. With a single-counter design, an asynchronous adder would be necessary to add an offset (ΔT) to each time stamp. In our implementation, a similar design is used for the two counters.

In our current implementation, the width of the counters, the comparator, and the incrementer is four bits. The queue is composed of 18 stages.

D. Analog/Digital Partitioning

To test and debug our system, we make the inputs and outputs of each segment accessible by partitioning the system. Although each segment can be implemented with a single custom analog/digital chip, we have partitioned the intrasegmental system, as shown in Fig. 5. This partitioning facilitates the testing and debugging at the intrasegmental level and decomposes the segment into smaller analog and digital sections.

The CPG chip contains two silicon Morris–Lecar neurons (i.e., burst-envelope neurons) [7], 32 synapses (16 per neuron), and an asynchronous decoder that interfaces the communications network with the synapses. Each synapse emulates a graded synaptic transmission that is described by a sharp and fast thresholding function. The AER chip contains an

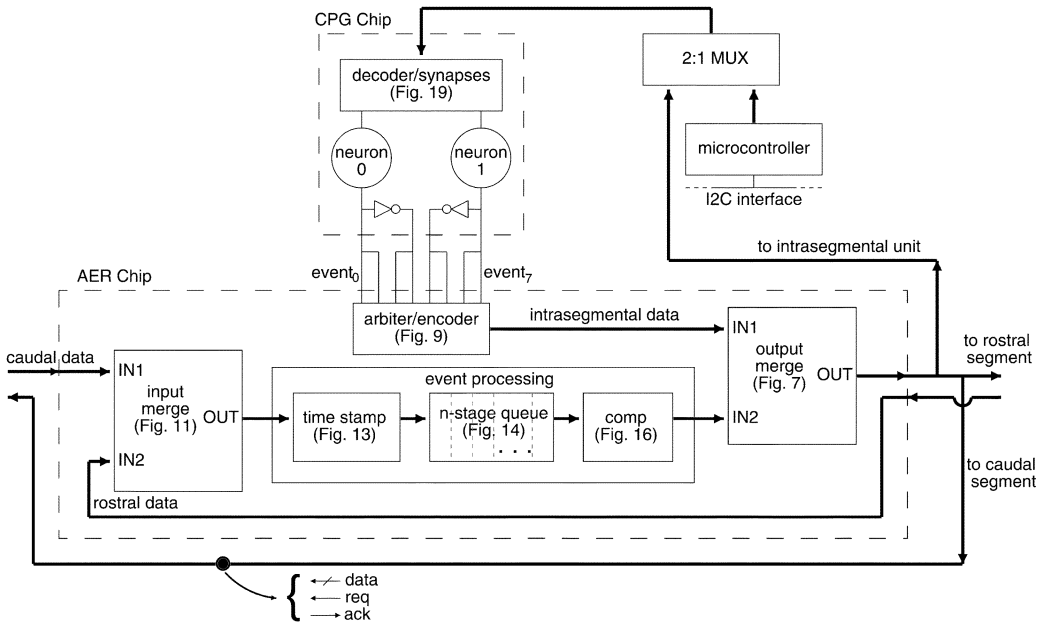

Fig. 5. Intrasegmental partitioning of analog pattern-generating circuit and digital communications network. Note that the output of the AER chip has three destinations.

eight-input arbiter/encoder section, an event processing section, and input and output merge sections. The arbiter/encoder section provides an interface between neurons and the communications network. This section detects events from neurons (rising edges), arbitrates between the events, and encodes the event types. It creates the appropriate handshaking signals that are compatible with the output merge section. The input merge section accepts events from neighboring segments, samples their a/d bit, and makes a decision whether to accept or drop the event. After the decision, it forward accepted events from both the rostral and caudal segments into the event processing section. As described in Section II-C, the event processing section time stamps, stores, and processes the events. The output merge section merges events arriving from the queue and the CPG chip and sends them to both the neighboring segments and the local CPG chip.

We use a microcontroller in each segment to program the analog synapses in the CPG chip and to deliver a programmable clock to the AER chip. A host processor delivers commands to the individual microcontrollers via an Inter-IC (I2C) interface.

Section III describes the communication between each of these circuits.

III. Asynchronous Methodology

Because biological systems operate asynchronously (the generation of action potentials between neurons is not synchronized), the choice to make the design asynchronous is another step toward mapping the architecture to the biological systems. Since segmented biological systems contain varied numbers of segments (from about 20 segments in the leech to about 100 in the lamprey), an asynchronous implementation facilitates the addition of segments without a major redesign and without having to redistribute global clock signals over a large number of segments.

A. Synthesis Methodology

We have utilized the synthesis methodology described by Martin [8]. Although many techniques for synthesizing asynchronous circuits exist, the following are the primary reasons for using this methodology.

- The rules for implementing processes are simple and intuitive, allowing for manual synthesis and optimization of the circuits.
- A high-level description or a communications program is decomposed into smaller processes, making the synthesis of complex modules tractable.
- The synthesis methodology yields primitives that directly translate to implementations at the transistor level, resulting in compact circuits suitable for VLSI implementation.
- This methodology has been demonstrated successfully in another neuromorphic system [4].

The first step in the compilation process is to write a high-level description of each process in the variant of CSP [9], known as communicating hardware processes (CHP), then each high-level description is decomposed into smaller processes, and then communication actions in each simplified process are expanded into a four-phase, handshake sequence. The elements in this sequence consist of wait statements and corresponding transitions of request and acknowledge lines of all the ports in the process. A production rule set is generated from this sequence for the design of pull-up and pull-down networks. To optimize the circuit implementation, signal transitions in the handshake-expansion sequence may be reshuffled as long as the four-phase handshake protocol is adhered to and program functionality remains unaltered. In addition, the predicates guarding each transition can be strengthened or weakened as long as they remain noninterfering (that is, the guards do not introduce signal contention problems).

We now briefly review some notation. To begin with:

- $\mathrm{signal}\uparrow \equiv$ drive signal to a high value;

- $\text{signal}\downarrow \equiv$ drive signal to a low value;
- $\text{action1}; \text{action2} \equiv$ sequential activity, complete action1 before beginning action2;
- $[\text{signal}] \equiv$ wait for signal to become TRUE.

In CHP, two processes sharing a common channel communicate with each other by using communications commands on their ports. The communications channel (and port) consists of a request line, an acknowledge line, and data lines. A high-level description of a program S is made up of multiple processes that are composed by one of three operators: the *sequential operator*, represented by a semicolon (;); the *concurrent* or *parallel operator*, represented by parallel bars ($\|$); and the *coincident operator*, represented by a bullet ($\bullet$). When processes are to be executed in parallel, as in

$$S \equiv S_1 \parallel S_2 \parallel \cdots \parallel S_n$$

each subprocess can be executed in any order. In

$$S \equiv S_1; S_2; \ldots; S_n$$

the subprocesses are executed in the order specified. When using a bullet operator, however, subprocesses are executed and completed at the same time. Note that the definition of the coincident operation is unambiguous if the subprocesses are independent or noninterfering (i.e., the processes do not share variables). Formally, the definition is that both subprocesses complete in the same state of the computation, where completion is defined as the point at which all possible continuations of the computation (traces) contain the remainder of the subprocess.

The execution of processes may be guarded by a Boolean predicate. When the program is designed such that only one guard is TRUE at any given time, the selection of the command is deterministic. The *deterministic selection operator* is a hollow bar ($[\!]$). The deterministic choice in

$$S \equiv [G_1 \rightarrow S_1 [\!] G_2 \rightarrow S_2 [\!] \ldots [\!] G_n \rightarrow S_n]$$

is feasible if and only if at most one guard evaluates to TRUE. When several guards may be TRUE at the same time, the selection of the command is nondeterministic. The *nondeterministic selection operator* is a single bar ($|$).

A program is repeated forever by using the *repetition operator*, as in $^*[S]$. A *probe* command at a given port, $\bar{A}$, is to check whether communication is pending. $\bar{A}$ can be considered a guarded expression that checks whether the request on port A is TRUE. A guarded command in closed brackets $[G]$ indicates "hold until G is TRUE." The statement $X?u; Y!u$ indicates that data on port X is to be read and stored in buffer u and subsequently output on port Y. In the situation in which buffering is unnecessary, one would write $Y!(X?)$.

B. Synthesis Example: The Merge Process

We use the MERGE process to demonstrate the synthesis methodology. We start with the definition of a circuit that performs a MERGE operation [8]. The circuit, whose block-level diagram is shown in Fig. 6(a), is used for merging events arriving from neighboring segments, at port L and at port R, and inserting into a stage (e.g., a queue) at port Q. The control signals of ports L and R contain request lines that are inputs, L_i and R_i, respectively, and acknowledge lines that are outputs,

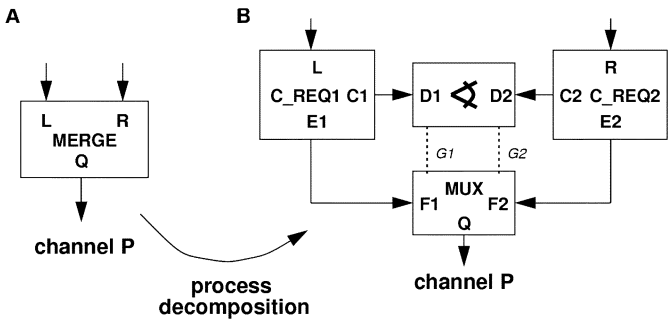

Fig. 6. (A) Block-level diagram of the MERGE process and (B) decomposition into smaller subprocesses.

L_o and R_o, respectively; port Q contains a request line that is an output, Q_o, and an acknowledge line that is an input, Q_i.[1]

To increase the throughput of the system, the data arriving at port L and port R may be buffered locally; however, because compactness, and not speed, is important in our application, we do not buffer the data. Thus, the high-level description is written as

$$\text{MERGE} \equiv {}^*\big[\big[\bar{L} \rightarrow Q!(L?) \mid \bar{R} \rightarrow Q!(R?)\big]\big] .$$

Because the events are asynchronous, a nondeterministic choice is necessary to gain access to the queue (channel P). In addition, as the data is not stored locally, a request at an input port should not be acknowledged until the subsequent request at port Q is acknowledged.

As shown in Fig. 6(b), this problem is made tractable by decomposing the MERGE process into the following four subprocesses: C_REQ1, C_REQ2, ARBITER, and MUX. Processes C_REQ1 and C_REQ2 make requests for gaining access to channel P and to coordinate communication between ports L, $C1$, and $E1$ and R, $C2$, and $E2$, respectively. Process ARBITER makes the nondeterministic choice and grants permission for the access of channel P. Process MUX multiplexes data from port $F1$ or port $F2$ onto port Q and initiates communication on port Q. The CHP description for these processes is as follows:

$$\begin{aligned}
\text{ARBITER} &\equiv {}^*\big[\big[\overline{D1} \rightarrow D1 \mid \overline{D2} \rightarrow D2\big]\big] \\
\text{C_REQ1} &\equiv {}^*\big[\big[\overline{L} \rightarrow C1 \bullet (E1!(L?))\big]\big] \\
\text{C_REQ2} &\equiv {}^*\big[\big[\overline{R} \rightarrow C2 \bullet (E2!(R?))\big]\big] \\
\text{MUX} &\equiv {}^*\big[\big[\overline{F1} \rightarrow Q!(F1?) [\!] \overline{F2} \rightarrow Q!(F2?)\big]\big] .
\end{aligned}$$

The role of C_REQ1 and C_REQ2 is understood by studying the sequence of atomic actions that occur if a request is received, for example, on port L (i.e., $L_i\uparrow$). In this situation, C_REQ1 will make a request to the arbiter ($C1_o\uparrow$); when the arbiter acknowledges ($[C1_i]$), C_REQ1 makes a request to the multiplexer ($E1_o\uparrow$); when the multiplexer acknowledges ($[E1_i]$), C_REQ1 releases control of the channel by dropping its request to the arbiter ($C1_o\downarrow$); C_REQ1 drops its request to the multiplexer ($E1_o\downarrow$); and finally, C_REQ1 acknowledges the original request at port L ($L_o\uparrow$). During this process, if a request were received at port R, the request would be blocked by the arbiter. To

[1]To maintain a consistent notation, we designate request, acknowledge, and data signals on a passive port, X, as X_i, X_o, and X_{data}, respectively. The request, acknowledge, and data signals on an active port are designated as X_o, X_i, and X_{data}, respectively.

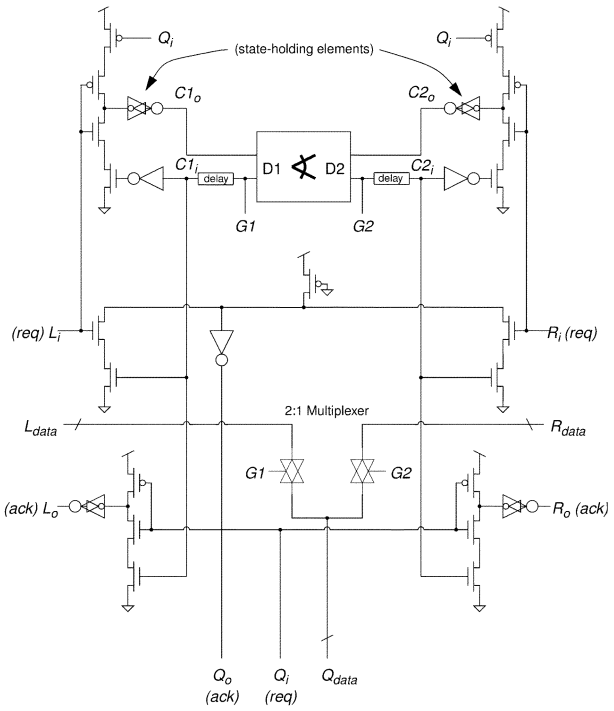

Fig. 7. Circuit implementation of the MERGE process. Note the symbols used for state-holding elements (staticizer circuits) and transmission gates (at output of Q_{data}). The gain of the feedback element of the staticizer is approximately 1/8 the gain of the driving circuit.

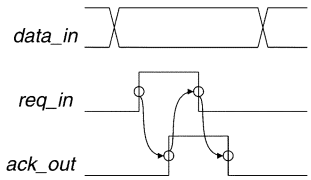

Fig. 8. Four-phase protocol and bundled data convention. The data input is captured (seen as a transition on $data\_in$) prior to the rising edge of req_in.

guarantee that the data on port Q remains stable, the acknowledge to the arbiter must be delayed until the acknowledge from the queue is received.

For C_REQ1 (and C_REQ2), although subprocesses $C1$ and $E1!(L?)$ ($C2$ and $E2!(R?)$) are interrelated (there is dependence through a variable), we use the coincident operator to signify that the two subprocesses are to be executed and completed at the same time. The ambiguity in the definition will be resolved when we perform handshake expansion on the program. Following the synthesis compilation process defined in [8], the resulting circuit is shown in Fig. 7.

C. Handshaking Protocol

We assume that data signals are valid when the control signal becomes valid (i.e., the signals have settled to their final values). Specifically, when a request line is raised by a preceding stage, the stage receiving the request assumes that the data on its data port is valid. This assumption is known as the bundled data convention [10]. It is illustrated in Fig. 8 using a four-phase handshaking protocol. To ensure that the timing assumption is valid, we have carefully engineered and analyzed the delays on data and handshaking signals. A data request is propagated to a subsequent stage following the rising acknowledgment of the capture of the data in the current stage. This corresponds to the narrow data release scheme [11].

IV. Implementation

In order to implement this architecture, the full processing of an event must be considered. First, we need to generate an event from the rapid rise or fall in the membrane potential of a spiking neuron. The event must then be arbitrated, encoded, and inserted into the communications channel. Next, the event must be transmitted to all of the other segments, and finally the event needs to be decoded such that it effects one synaptic module. The details of these event handling procedures are presented in this section.

A. Event Generation, Encoding, and Arbitration

Events in this communication architecture are generated by silicon neurons. Many such neurons have been presented in the literature [12]–[14], and these neurons generate either spike trains (i.e., action potentials) or burst envelopes that represent these spike trains that are encoded as events. CPG circuits are created from a set of these neurons. For our system, the neurons generate burst envelopes and are given by high voltages during spiking and low voltages during silence. As a result, two event types are encoded—one by the rising edge of an envelope (i.e., the beginning of a spike train) and the other by the falling edge of an envelope (i.e., the end of a spike train).

To interface neurons to asynchronous circuits, we must transmit each event with a pair of request/acknowledge lines that adhere to the four-phase handshake protocol that we have adopted. Note that there is an implicit timing assumption in this specification which is required for synthesis. Specifically, there must be sufficient delay between the rising and falling events in order for the communication action to occur and to be able to discern between both events; this is ensured because the periodicity of the events is on biological time scales (i.e., on the order of milliseconds).

Once an event is generated by a neuron in the CPG chip, the event must be inserted into the communications channel. The arbiter/encoder section of the AER chip accomplishes this task by arbitrating, encoding, and inserting the events into the communications channel. The decomposition of this process is shown in Fig. 9.

The actual insertion of events into the communications channel is controlled by the interaction between the arbiter tree and the NA (neuron/arbiter) modules, as shown in Fig. 9. The high-level description of the NA and the arbiter processes are

$$\mathrm{ARBITER} \equiv {}^*\big[\big[\overline{A_0} \rightarrow A_0 \mid \cdots \mid \overline{A_7} \rightarrow A_7\big]\big]$$
$$\mathrm{NA}_i \equiv {}^*\big[\big[\overline{A_i} \rightarrow B_i \bullet C(!); A_i\big]\big] .$$

When an event is received, the corresponding NA module makes a request to an asynchronous arbiter for access to the communications channel. The temporal ordering of multiple events is preserved by the arbiter, and in the case of nearly simultaneous events, a nondeterministic[2] choice is made. After receiving access, the NA module is allowed to make a request

[2]The choice, however, may be biased because mismatches in the arbiter might cause some inputs to be favored.

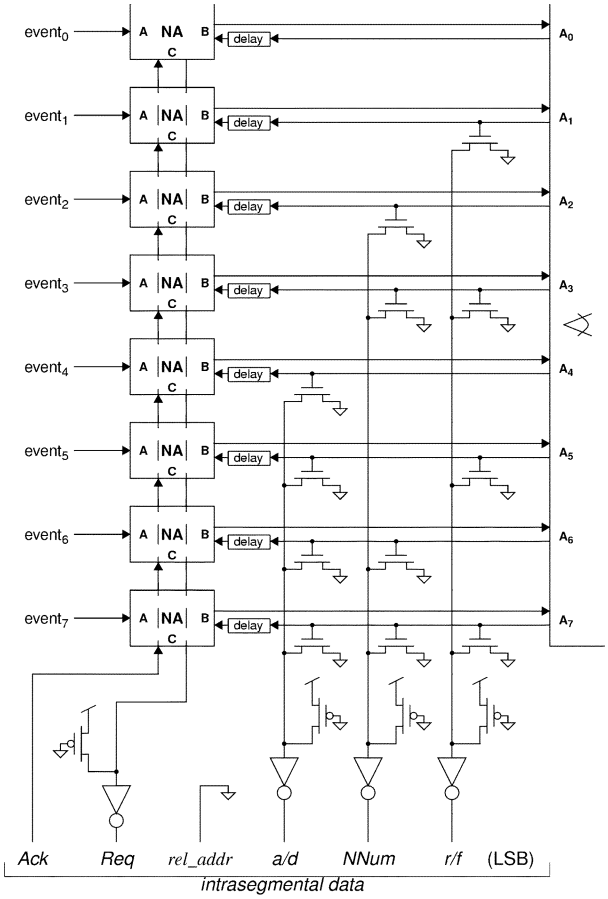

Fig. 9. Arbiter/encoder section illustrating arbitration and encoding of events.

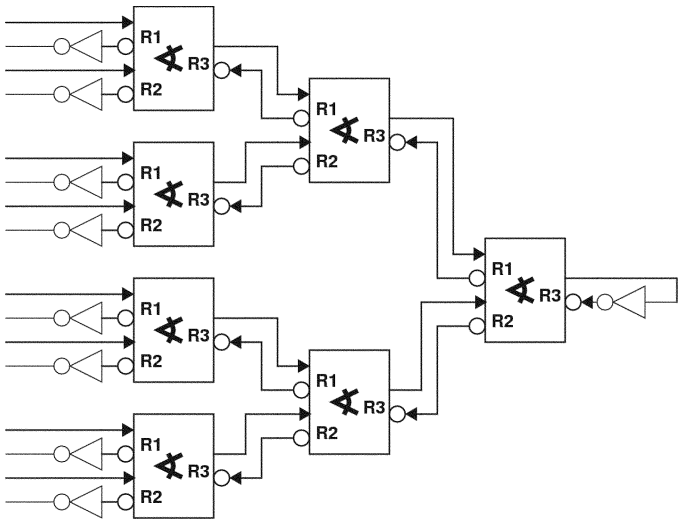

Fig. 10. Eight-input arbiter used in the arbiter/encoder section.

to the communications channel. As a result of the bundled data convention, however, the request is not made until the data on the output port is valid. This condition, fulfilled by ensuring that the delay in the control path is greater than the delay in the data path, is guaranteed by adding sufficient delay in the acknowledge lines of the arbiter (see Fig. 9). The bullet operator in the above description signifies that the two subprocesses complete in the same state. The implementation of the arbiter/encoder section is similar to the n-input merge; however, instead of multiplexing data, we are creating data. Because the acknowledge lines of the arbiter are mutually exclusive, the event address is encoded by using these lines. In addition, the mutually exclusive acknowledge lines facilitate the use of a common acknowledge line at port C of the NA modules. We use a wired-OR configuration for the generation of an output request.

We implement an n-input arbiter with a tree of two-input arbiters. The tree, whose depth is $\log_2(n)$, contains $n-1$ arbiter cells. The eight-input arbiter used in the arbiter/encoder section is shown in Fig. 10. The high-level description of an individual arbiter cell, containing two input ports, $R1$ and $R2$, and a single output port, $R3$, is given as

$$\text{arb\_cell} \equiv {}^{*}\left[\left[\overline{R1} \rightarrow R3; R1 \mid \overline{R2} \rightarrow R3; R2\right]\right].$$

The implementation of the arbiter cell is based on a previously published design [15]. (Note: To adhere to the four-phase protocol, the original design has since been improved [16].)

To complete the arbitration process, the acknowledge signal on top of the tree is generated from the output request of the last stage. Because the acknowledge signals are active low, an inverted version of the output request is connected to the acknowledge line.

Because a neuron's axon projects to both sides of the segment, every rise and fall in the neuron's membrane potential should generate two events: an ascending event and a descending event. Two events are produced by connecting the neuron's output to two handshake HS modules, which in turn, connect to two inputs of the arbiter/encoder section.

B. Event Transmission

Once the events are generated by the individual neurons and inserted into the communications network, they must be transmitted to all other segments. Events arriving from ascending and descending segments are filtered and then merged by the input merge section, and subsequently, inserted into the event-processing section. In the event-processing section, the events are time stamped, stored in a queue for a length of time corresponding to the intersegmental delay, and their addresses are incremented as the events exit the queue. The events are then merged with events arriving from the CPG chip by the output merge section. After the events exit the output merge section, the events are transmitted to the neighboring segments and the local segmental unit.

1) Merging and Time-Stamping Events: The output merge section simply merges two event streams (without processing the events). The input merge section, however, has the additional task of dropping the circulating events. This task is accomplished by checking the polarity of the direction bit in the event data. Descending events $(a/d = 0)$ arriving from the rostral segment and ascending events $(a/d = 1)$ arriving from the caudal segment are allowed to pass. In contrast, ascending events arriving from the rostral segment and descending events arriving from the caudal segment are rejected. The process decomposition of the input merge section is shown in Fig. 11. The events are filtered by the following two subprocesses:

$$\text{DES\_DIR} \equiv {}^{*}\left[\left[\overline{A} \rightarrow \left[\frac{a}{d} \rightarrow B; A[\!]\neg\frac{a}{d} \rightarrow A\right]\right]\right]$$

$$\text{ASC\_DIR} \equiv {}^{*}\left[\left[\overline{A} \rightarrow \left[\neg\frac{a}{d} \rightarrow B; A[\!]\frac{a}{d} \rightarrow A\right]\right]\right].$$

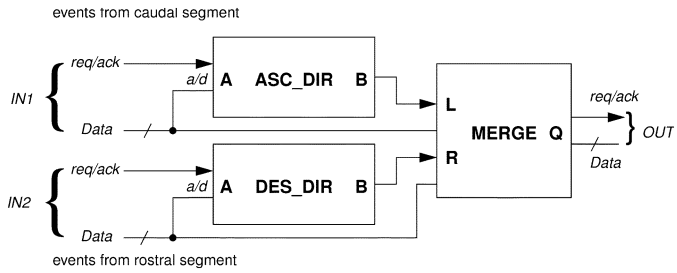

Fig. 11. Process decomposition of the input merge section. The direction of events is checked by ASC_DIR and DES_DIR.

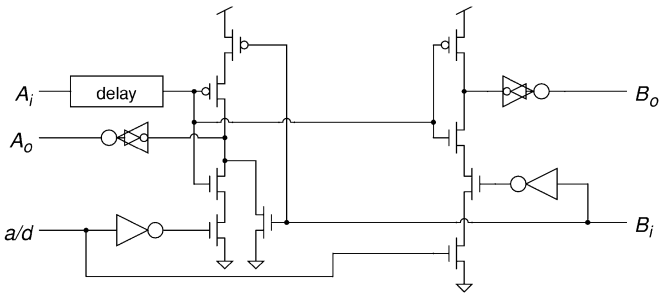

Fig. 12. Circuit implementation for DES_DIR.

The handshake expansion of DES_DIR is

$$*\left[\left[A_i \wedge a/d \rightarrow B_o\uparrow; [B_i]; A_o\uparrow, B_o\downarrow; [\neg B_i]; [\neg A_i]; A_o\downarrow \right.\right.$$
$$\left.\left. [\![A_i \wedge \neg\frac{a}{d} \rightarrow A_o\uparrow; [\neg A_i]; A_o\downarrow\right]\right].$$

The production rule set obtained from the above expansion is

$$A_i \wedge a/d \wedge \neg B_i \rightarrow B_o\uparrow \qquad (A_i \wedge \neg a/d) \vee B_i \rightarrow A_o\uparrow$$
$$\neg A_i \rightarrow B_o\downarrow \qquad \neg A_i \wedge \neg B_i \rightarrow A_o\downarrow .$$

Using this production rule set, the circuit schematic of DES_DIR is shown in Fig. 12. Because the input events travel across chip boundaries, to guarantee that our bundled-data assumption holds, we add delay to the input request, A_i. The implementation of ASC_DIR is similar to that of DES_DIR; however, the polarity of the a/d bit is opposite to that of in DES_DIR.

The first step in processing the events is appending a time stamp to the event data. The output of a synchronous counter is used as the time stamp. Because a request from the input merge section and an upward transition in ∼CLOCK may occur within an arbitrarily small interval, a nondeterministic choice is necessary. Thus, the high-level description of the time stamp process is given as

$$\text{TIME\_STAMP} \equiv *[[\sim\text{CLOCK} \rightarrow (t1 := t1 + 1)$$
$$B_{\text{data}}\langle 10:7\rangle := t1 \mid \overline{A} \rightarrow B!(A?)]]$$

where events arrive at port A and exit at port B. The above process delays a transaction on the output port of TIME_STAMP until the counter has finished updating, or delaying the clock for the counter until the transaction on the output port is finished. In the latter case, the assumption is that the delay will be small relative to the period of ∼ CLOCK. If the delay were large relative to the period of ∼ CLOCK, the time stamp would drop a clock tick, affecting only items already in the queue since the delay is measured relative to that register. In practice, this never occurs since the clock period is very large relative to electronic delays.

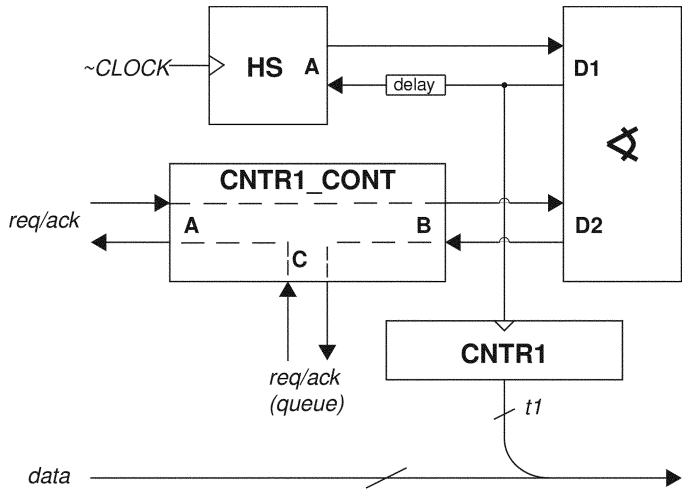

Fig. 13. Process decomposition of the TIME_STAMP process.

The process decomposition of the TIME_STAMP process is shown in Fig. 13. The clock signal, which is an internally buffered signal, is converted to a pair of handshake signals by the HS module. To satisfy the bundled-data assumption (that is, to ensure that the output of the counter is valid before sampling its contents), we add delay to the acknowledge signal on port A of the HS module. The undelayed version of this acknowledge signal is used to update the counter (on every rising edge). Subprocess CNTR1_CONT controls communication among the input port (port A), the arbiter (port B), and the queue (port C). Its high-level description is

$$\text{CNTR1\_CONT} \equiv *[[\overline{A} \rightarrow B \bullet C!(A?)]].$$

The handshake expansion can be written as

$$*[[A_i]; B_o\uparrow; [B_i]; C_o\uparrow; [C_i]; A_o\uparrow;$$
$$[\neg A_i]; B_o\downarrow; [\neg B_i]; C_o\downarrow; [\neg C_i]; A_o\downarrow].$$

The production rule set for the above expansion yields three wires:

$$A_i \rightarrow B_o\uparrow \quad B_i \rightarrow C_o\uparrow \quad C_i \rightarrow A_o\uparrow$$
$$\neg A_i \rightarrow B_o\downarrow \quad \neg B_i \rightarrow C_o\downarrow \quad \neg C_i \rightarrow A_o\downarrow .$$

The wires are shown as dashed lines in Fig. 13.

2) Sorted Queue: Within a single communication channel, events may arrive at the QUEUE out of temporal order. Thus, a reordering of events in the QUEUE is necessary. Although not implemented with the architecture described in this paper, a bubble sort algorithm is sufficient to maintain proper ordering: each event progresses through the queue, overtaking events ahead of it, until its time stamp is greater than the time stamp of the event in the stage ahead of it [6].

3) Storing Events—Asynchronous FIFO: A single stage of the queue is shown in Fig. 14. To minimize area, we use latches as the basic storage element in the register. In our implementation, the latches are transparent and state holding for low and high levels of the clock, respectively. When the queue stage is empty, the latches are transparent and ready to capture the input data. After the input request is raised, the data is captured by the latches. The storage and transmission of data are

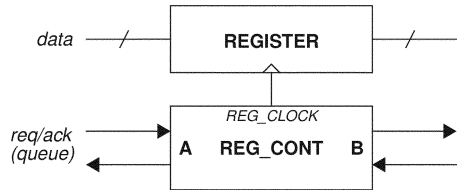

Fig. 14. Single stage of the queue.

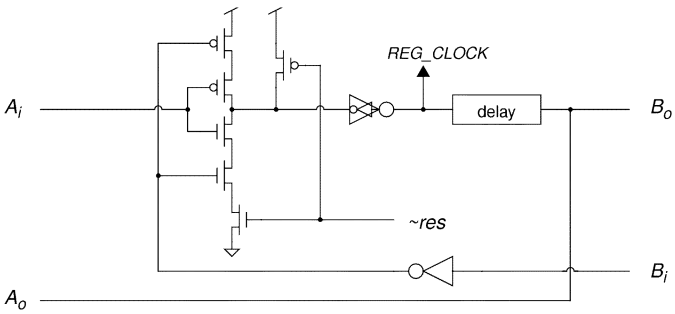

Fig. 15. Implementation of REG_CONT.

controlled by process REG_CONT. The design is straightforward and has been previously described in [10]. Its implementation is shown in Fig. 15. Note that in [10], the circuit is used to control a two-phase pipeline, but we used the same design to control a four-phase pipeline. Also note that the maximum throughput with this pipeline control is about one half of that achievable with more complex designs because adjacent stages cannot be concurrently active in the steady state. Since we are using the queue to delay events on biological time scales, this consequence is not detrimental to our design, and as a result, the queue throughput is sufficient to move all events forward one place during one inter-arrival time. In addition, because the data validity scheme is narrow (early release), events propagate forward in the queue at the maximum rate corresponding to the delay of one C-element, providing no preceding events exist in the queue. This ensures that head-of-line events reach the head of the queue as early as possible, minimizing the probability of queue-induced jitter in the desired delay.

4) Delaying Events: At the tail of the queue, the time stamp of an emerging event $(t1)$ is compared with the output of a second counter $(t2)$. When $t1 = t2$, the event is sent to the incrementer (INCR), where its relative address in incremented. Thus, the intersegmental delay is determined by the offset between the two counters.

The process that receives an event and compares its time stamp with the output of the second counter is described as

$$\mathrm{COMP} \equiv {}^*[[\overline{\mathrm{A}} \rightarrow [\mathrm{G_{t1=t2}} \rightarrow \mathrm{B!(A?)} \| \mathrm{G_{t1 \neq t2}} \rightarrow \mathrm{skip}]]]$$

where events are received at port A and exit at port B. Also, the guard $G_{t1=t2}$ is a Boolean-valued comparison operation obtained by sampling the event time stamp and the output of the counter; its value is TRUE when $t1 = t2$ and is FALSE otherwise. The *skip operation*, which is read as "do nothing," prevents the process from completing until the guard $G_{t1=t2}$ is TRUE. As illustrated in Fig. 16, we decompose the COMP process into two subprocesses, COMP_REQ and DO_COMP. The latter subprocess, which contains the second counter, does the comparison and completes the communications process when $t1 = t2$. COMP_REQ makes a request for a comparison and coordinates the communication between the queue and the incrementer. The high-level descriptions of the subprocesses are

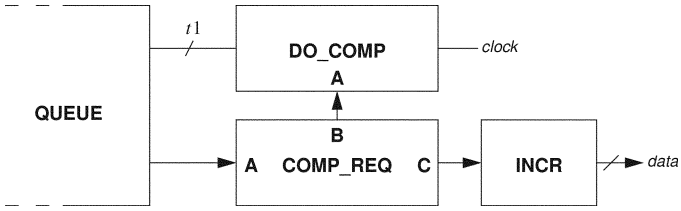

Fig. 16. Events exit the queue through the control of COMP_REQ and DO_COMP.

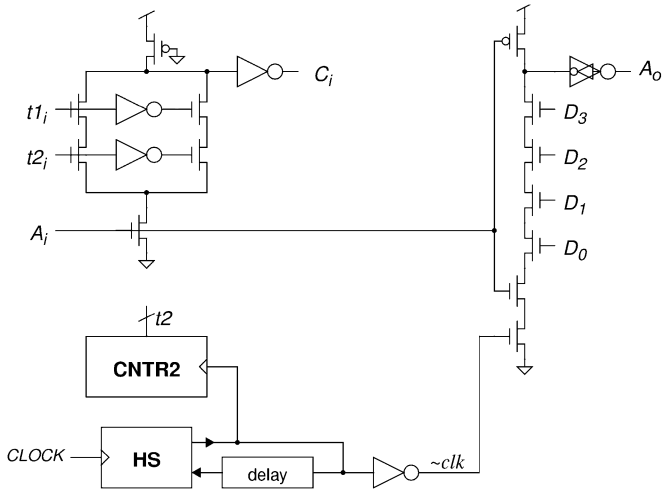

Fig. 17. Implementation of the DO_COMP process.

$$\mathrm{COMP\_REQ} \equiv {}^*[[\overline{A} \rightarrow B; C!(A?)]]$$
$$\mathrm{DO\_COMP} \equiv {}^*[[\overline{A} \rightarrow [G_{t1=t2} \rightarrow A \| G_{t1 \neq t2} \rightarrow \mathrm{skip}]]].$$

The handshake expansion of DO_COMP is

$${}^*[[A_i]; [G_{t1=t2}]; A_o\uparrow; [\neg A_i]; A_o\downarrow].$$

The production rule set is

$$A_i \wedge G_{t1=t2} \rightarrow A_o\uparrow$$
$$\neg A_i \rightarrow A_o\downarrow .$$

Note that the sampling should occur every time an input request arrives and each time the counter is updated. However, to prevent erroneous output during the update of the counter, the comparison is disabled at the onset of the rising edge of the clock for a sufficient time needed for the output of the counter to settle. Circuit implementation of DO_COMP is shown in Fig. 17. The signal used to disable the comparison during the update of the counter, $\sim \mathrm{clk}$, is obtained from the handshake module. This signal is low at the onset of CLOCK and remains low for sufficient time to allow the outputs of the counter to become valid. The guard that expresses the requirement that $t1 = t2$ is

$$G_{t1=t2} = \sim\!\mathrm{clk} \wedge D_0 \wedge D_1 \wedge D_2 \wedge D_3$$

where

$$D_i = [(t1_i \wedge t2_i) \vee (\neg t1_i \wedge \neg t2_i)] \wedge A_i.$$

The handshake expansion of COMP_REQ is

$${}^*[[A_i]; B_o\uparrow, [B_i]; C_o\uparrow; [C_i]; A_o\uparrow;$$
$$[\neg A_i]; B_o\downarrow, C_o\downarrow; [\neg B_i]; [\neg C_i]; A_o\downarrow].$$

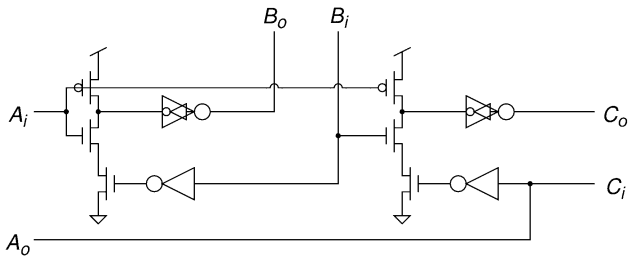

Fig. 18. Implementation of the COMP_REQ process.

To simplify the implementation, we have shuffled the atomic actions. Specifically, $A_o\downarrow$ is the last atomic action so that A_o can be implemented with a single wire. The production rule set is

$$C_i \to A_o\uparrow \qquad A_i \wedge \neg B_i \to B_o\uparrow \qquad B_i \wedge \neg C_i \to C_o\uparrow$$
$$\neg C_i \to A_o\downarrow \qquad \neg A_i \to B_o\downarrow \qquad \neg A_i \to C_o\downarrow .$$

The implementation is shown in Fig. 18.

The final process in the event-processing section is responsible for incrementing the relative address. Its process description is given as

$$\mathrm{INCR} \equiv {}^*[[\overline{A} \to (A?u) \bullet (B!(u+1))]]$$

where events enter at port A and exit at port B. An intermediate register, u, is used to increment the relative address. For the implementation of the above process, we use the same circuit used for the REG_CONT process (a c-element). However, because of the additional processing, we use edge-triggered storage devices with appropriate combinational logic at their inputs instead of latches.

5) Output Events: Events emerging from INCR and the arbiter/encoder section are merged and sent, via the output merge section, to the neighboring AER chips and the local CPG chip. Because the intersegmental delay (ΔT) is much greater than the time necessary to transmit the events across the segments (δT), we do not store (or pipeline) the events after they exit the output merge section. Because the output merge section sends events to three passive ports, we use a wired-and configuration for the output acknowledge signal. This connection, reduces the pad count, but at the cost of a steady-state current.

C. Event Reception

Each event is represented by an address that encodes the location of its origin relative to its current position in the system, the neuron in the CPG from which it was generated, and whether the event is a rising-edge or falling-edge event. As events enter the CPG chip, they are decoded such that they affect one synaptic module in the CPG chip. The role of each synaptic module is to appropriately excite or inhibit the CPG neurons. If the synaptic weight is configured positive, it will excite the neuron in the CPG; if it is configured negative, it will inhibit the neuron. In our prototype system, all the synaptic weights are inhibitory. Each synaptic module contains a floating-gate transistor for storing an analog weight, and the weights are programmed using Fowler–Nordheim tunneling and hot-electron injection [17], [18]. Although every neuron in the system is connected to every other neuron, connections between neurons can effectively be broken by programming a synaptic weight of zero.

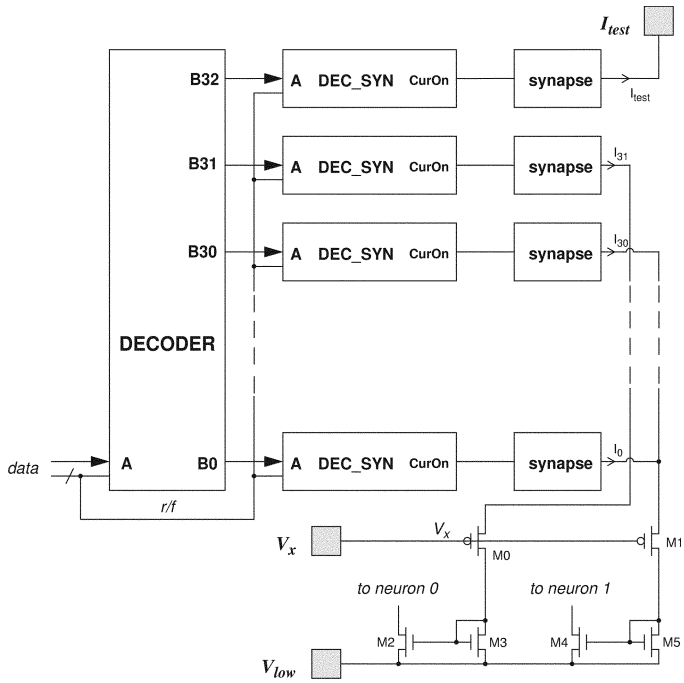

Fig. 19. Decoding event data and establishing inter- and intrasegmental connections.

The interface between the communications channel and the synapses of the neurons is shown in Fig. 19. The interface contains a decoder and intermediate modules (DEC_SYN) that interface the decoder to the synaptic modules. The synaptic modules use the CurOn signals, which are generated in the respective DEC_SYN modules, to turn on or to turn off the inhibitory currents. The DEC_SYN module generates the CurOn signal by sampling the r/f bit of the event data. The description of the decoder is given as

$$\mathrm{DECODER} \equiv {}^*[[\overline{A} \to [G0 \to B0 [] \ldots [] G32 \to B32]; A]]$$

where events are received at port A and are directed to one of 33 ports, $B0$–$B32$, depending on the mutually exclusive guards, $G0$–$G32$. The handshake expansion below illustrates how we implement two rows of the decoder:

$$\begin{aligned} &{}^*[[G1 \wedge A_i]; B1_o\uparrow; [B1_i]; A_o\uparrow; \\ &\quad [\neg A_i]; B1_o\downarrow; [\neg B1_i]; A_o\downarrow] \\ &\cdots \\ &{}^*[[G32 \wedge A_i]; B32_o\uparrow; [B32_i]; A_o\uparrow; \\ &\quad [\neg A_i]; B32_o\downarrow; [\neg B32_i]; A_o\downarrow]. \end{aligned}$$

Because of the bundled-data convention, the guards are obtained by sampling the event address at the same time the input request arrives. The Boolean variables Gi_1–Gi_6 represent inverted or noninverted versions of the individual bits of the event address depending on the address to be decoded by row i. Because the guards are mutually exclusive and only a single DEC_SYN module is active at any given time, we use a wired-OR connection for the acknowledge line at port A.

The DEC_SYN module, whose sole purpose is to generate the CurOn signal, is described as follows:

$$\mathrm{DEC\_SYN} \equiv {}^*\left[\left[\overline{A} \to [r/f \to \mathrm{CurOn}\uparrow \,[]\, \neg r/f \to \mathrm{CurOn}\downarrow]\,; A\right]\right].$$

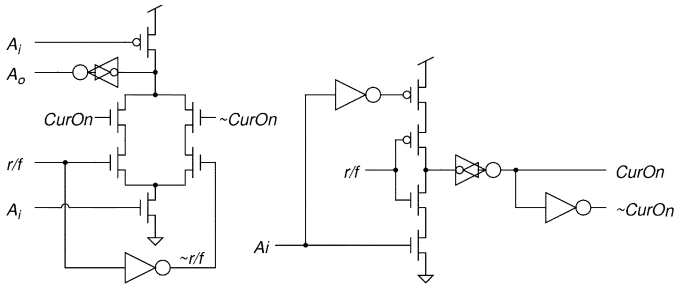

Fig. 20. Circuit implementation of the DEC_SYN module.

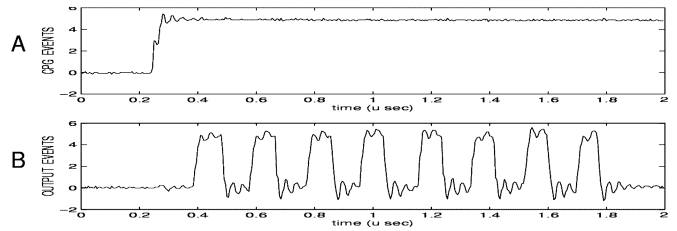

Fig. 21. Scope trace showing output events in a single communication node when the node is injected with eight simultaneous events.

The handshake expansion is

$$*\Big[\Big[\Big[\frac{r}{f}\wedge A_i\Big];\mathrm{CurOn}\uparrow;[\mathrm{CurOn}];A_o\uparrow;[\neg A_i];A_o\downarrow\ \Big[\!\Big]\ \Big[\neg\frac{r}{f}\wedge A_i\Big];\mathrm{CurOn}\downarrow;[\neg\mathrm{CurOn}];A_o\uparrow;[\neg A_i];A_o\downarrow\Big]\Big].$$

The circuit implementation is shown in Fig. 20. After the completion of a communications process, the state of the CurOn variable is held to the appropriate value (depending on r/f) until the start of another communications process. This facilitates the implementation of graded synaptic transmission.

V. System Analysis

In order to understand the functionality and the scalability of our system, we have looked at three critical issues: the channel capacity, the queue size, and the synaptic connections. We analyzed the capacity of the channel as well as the size of the queue in order to support data rates that we expected. We also considered the scalability of this architecture to larger systems through the addition of synaptic connections.

A. Channel Capacity

The speed of the channel is limited by the interchip communication as a result of the wired-AND connection (introduced in Section IV-B5). Therefore, in order to measure the channel capacity, we need to measure the maximum event rate between the local CPG network and the neighboring segments. The events are generated by applying a step input to the inputs of the arbiter/encoder section. When events are generated, they propagate through the arbiter/encoder section and the output merge section and are then received by the neighboring segments and the local CPG chip. The rate of the output requests is an approximate measure of the channel capacity. Fig. 21 shows a scope trace of the output request line when a step input is applied to the eight pads that receive events from the local CPG network. The channel capacity, which is primarily limited by pad capacitance and drivers, is shown in the figure to be approximately eight output events in 1.6 μs, or 5×10^6 events per second.

B. Queue Size and Scaling

The queue in our N-segment lamprey system stores events for a fixed time corresponding to the intersegmental delay ΔT, as defined by

$$\Delta T = T/N \tag{1}$$

where T is the time required for an event to travel the entire length of the system.

Assuming the events are uniformly distributed and the system is in stochastic equilibrium (the average input rate into the communications network is equal to the average output rate), the total number of events in the system Ω is given by

$$\Omega = \lambda \cdot N \cdot T \tag{2}$$

where $\lambda = f \cdot n$ is the average event rate per segment, f is the lamprey swim frequency, and n is the number of events per swim cycle. In our case, each segment in our burst-envelope implementation will output four $(n = 4)$ events per swim cycle: one event for each rising and falling edge for each of the two neurons in the segment.[3] In (2), the total number of events generated per second is given by the product of the first two terms, $\lambda \cdot N$, and T gives the average time that an event resides in the queue. The total number of events in a single segment is, therefore, given by

$$\omega = \lambda \cdot T. \tag{3}$$

If we use this architecture to create a compartmental model of the locomotion system of an entire animal, each compartment (segment in our model) represents a $1/N$ portion of the animal's body. The total time delay T through the system represents the axonal time delay down the total length of the animal's body, and, thus, is a constant independent of N; in other words, ΔT is inversely proportional to N as described by (1). Given that the total number of events in the queue, ω, sets the necessary size of the segmental queue, this queue size is also independent of N and is a function only of f, T, and n as described by (3).

In an intact lamprey, the swim frequency f can vary from 0.25 to 10 Hz [19], thus generating an event rate λ of 1–40 events per second per segment in each swim cycle (for $n = 4$), and the total axonal time delay T can vary from 0.1 to 4.0 s. Therefore, the maximum queue size $\omega_{\max}$ in the worst case scenario for our burst-envelope implementation is given by the largest swim frequency $f_{\max}$ and the longest total delay $T_{\max}$, as follows:

$$\omega_{\max} = f_{\max} \cdot n \cdot T_{\max} = 10 \cdot 4 \cdot 4 = 160. \tag{4}$$

This worst case represents the largest event rate and the case in which the events are stored for the longest possible time in the queue.

In order to reduce the cost of silicon for a prototype system, we assumed a slightly more limited, but reasonable, swim

[3]A rising (or falling) edge actually generates two events in the queue—one in the ascending direction and one in the descending direction. These two events are each stored in the queue for an average time of $T/2$, resulting in a combined average time of T.

frequency of 2 Hz and a maximum total delay of 2 s. Using these numbers, we would need 16 queue elements per segment to handle the resulting expected number of events. We built a queue with 18 elements per segment in order to demonstrate a proof-of-concept for our system. A queue of this size will enable us to model both juvenile lampreys (larger f, smaller T) and adult lampreys (smaller f, larger T).

The assumptions and implementations (spiking or burst envelopes) under which we are working affect the size of the queue. The following three variances on the assumptions are particularly important.

1) If the events arriving at a particular segment are not uniformly distributed and actually arrive in bursts, then more queue stages would be required to faithfully preserve the spatiotemporal patterns in the system.
2) In a spiking-neuron implementation, the segmental event-generation rate λ is equal to the average spike rate per neuron multiplied by the number of neurons. This value—which would likely be much larger than that for a burst-envelope implementation—would be independent of the swim frequency of the animal f and, thus, would change the system assumptions.
3) If only a portion of the animal's body were being modeled while holding constant the intersegmental delay ΔT, the total number of events to be stored in a single segment ω would vary proportionally with the number of segments N.

C. Synaptic Connections

The intersegmental architecture that we have developed is inherently modular, making the resulting system easily scalable through the addition or subtraction of segments. Although having no impact on the queue size, changes in the number of segments cause changes in the number of synapses per segment s and is given by

$$s = \alpha \cdot M^2 \cdot N \tag{5}$$

where α is the percentage of the number of segments down the length of the body in which synaptic connections are made and M is the number of neurons per segment; in our case, $M = 2$. Therefore, complete connectivity ($\alpha = 1$) results in M^2N synaptic connections. Note that the number of synapses per segment increases linearly with the number of segments.

In our present system, we implemented 16 segments ($N = 16$), two neurons per segment ($M = 2$), 32 synapses per segment ($s = 32$, $\alpha = 0.5$), and a queue size of 18 events per segment ($\omega = 18$). The size of an individual synapse is approximately 30 μm$\times$ 800 μm, or less than 0.5% of the total CPG-chip area. Thus, to implement a fourfold increase in the number of segments (to 64 segments), we would need to increase the size of the CPG chip by approximately a factor of two to accommodate the additional synapses.

VI. System Verification

We have designed and successfully tested a prototype communication system fabricated in a 2-μm process that is adequate to model a system of 16 segments with uniform axonal delay and a queue size of 18 events/segment. Each segment includes an AER chip and a CPG chip that contains the silicon neurons and synapses. The layout of the AER chip, shown in Fig. 22, contains an eight-input arbiter/encoder section, an event processing section, input and output merge sections, and an asynchronous decoder that interfaces the communications network with the synapses. Fig. 23 shows a part of the hardware implementation of the prototype system. Although intrasegmental oscillators with spiking neurons can be utilized with the communication network, we chose to model only slow dynamics of the bursting neurons through the implementation of the Morris–Lecar model of a neuron (i.e., burst-envelope neurons) [7], [13].

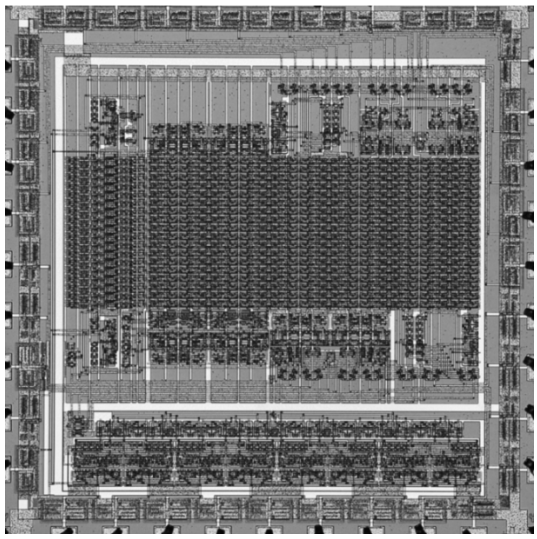

Fig. 22. Photomicrograph of the communication network chip.

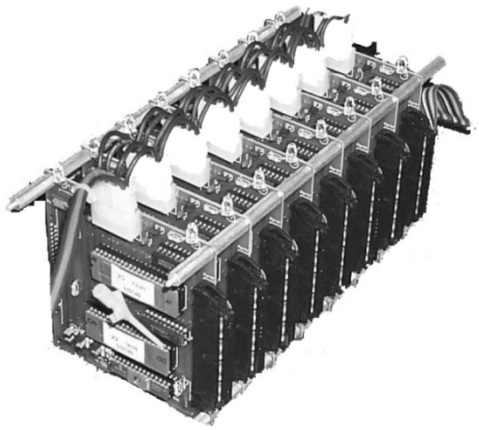

Fig. 23. Eight segments of the prototype system.

A. Component Operation

To test the input merge section, we monitored the events and their direction bits in three consecutive segments (Seg0, Seg1, Seg2). Fig. 24 shows the request signals that exit from the output merge section of each segment (Seg0_Req, Seg1_Req, Seg2_Req), their corresponding relative addresses (Seg0Addr, Seg1Addr, Seg2Addr), and their corresponding direction bits (Seg0_a/d, Seg1_a/d, Seg2_a/d). We observe that the temporal patterns displayed by the events are preserved in the ascending direction (from Seg2 to Seg0). Because the direction bits at each segment are $a/d = 1$, the observed pattern is consistent with our design—the events should propagate in

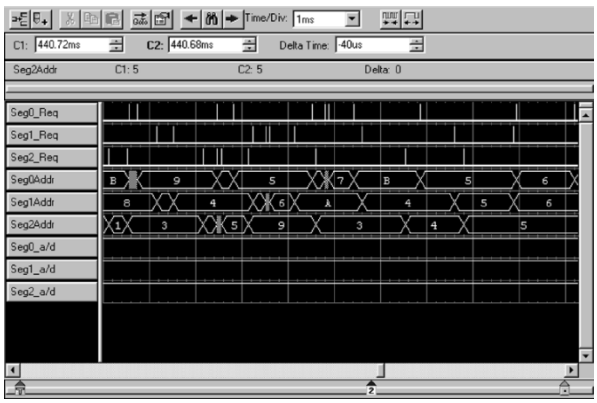

Fig. 24. Logic analyzer test results of a communication module—input merge section.

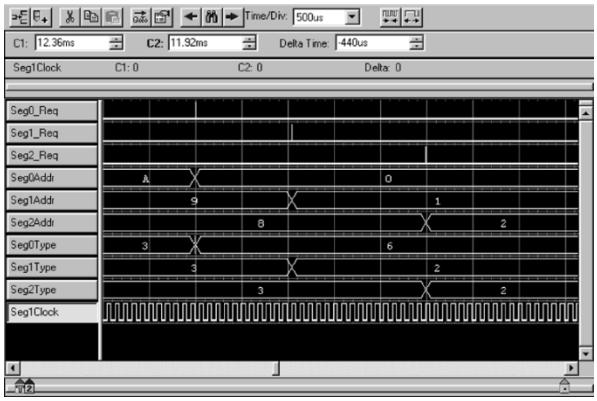

Fig. 25. Logic analyzer test results of a communication module—the portion of the event-processing section that is responsible for time stamping, storing, and delaying events.

the ascending direction when $a/d = 1$. The events do not propagate in the descending direction, demonstrating that the input merge section of each segment is effectively filtering the events. Fig. 24 also demonstrates that the events are delayed and their relative addresses are incremented at each segment.

To test the portion of the event-processing section that is responsible for time stamping, storing, and delaying events, we monitored an event as it entered the input merge section and counted the number of clock cycles before the event exits at the output merge section. Fig. 25 shows an event that originated in $\mathrm{Seg0}$ and then propagated to $\mathrm{Seg1}$ and $\mathrm{Seg2}$. The event is of descending type $(a/d = 0)$ and it starts with a relative address of zero (at $\mathrm{Seg0}$). The event resided in the queue of $\mathrm{Seg1}$ for the appropriate number of clock cycles—13 rising edge transitions occurred in $\mathrm{Seg1Clock}$ between the input request $(\mathrm{Seg0\_Req})$ and the output request $(\mathrm{Seg1\_Req})$.

B. System Operation

To demonstrate the system behavior without intersegmental connections, we show in Fig. 26 the autocorrelations of a neuron in two consecutive individual segments ($\mathrm{Seg4}$, $\mathrm{Seg5}$). The values of the autocorrelations are significant and are only present at the times corresponding to the harmonic frequencies of the burst envelopes of the neurons. Because the synaptic weights are set equal to zero, the synaptic connections should not have an effect on system behavior, which is evident from the figure.

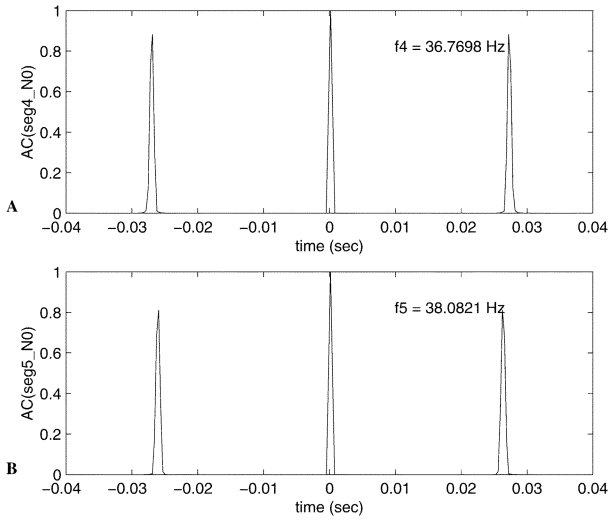

Fig. 26. Autocorrelation of a single neuron in (A) segment four and (B) segment five.

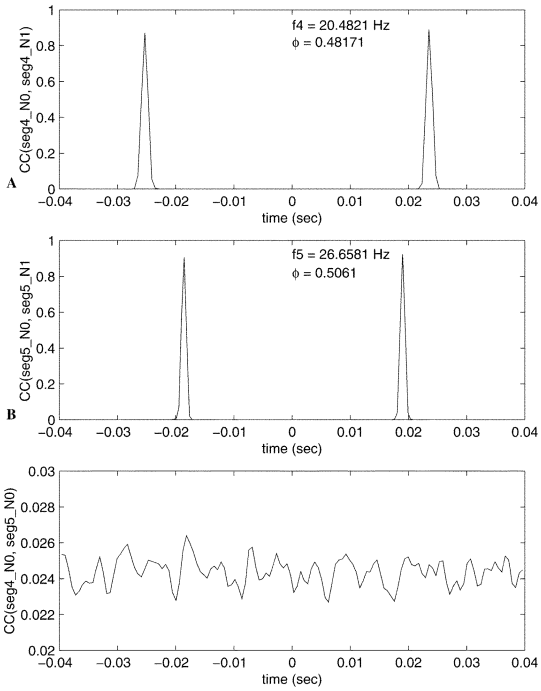

Fig. 27. Cross correlation between contralateral neurons in (A) segment four and (B) segment five and (C) between homolog neurons in segment four and segment five.

In another demonstration of the reliability of the communication system without intersegmental connections, we show in Fig. 27 cross correlations between contralateral neurons in two consecutive segments ($\mathrm{Seg4}$, $\mathrm{Seg5}$) and homolog neurons in these same two segments. The correlations are computed on periodic signals that are approximately 400 cycles long. This result shows that for reasonable time intervals, the reliability of the communications system is good. Fig. 27(c) also indicates, as expected, that synchronization did not occur when no intersegmental connections were in place; instead, only noise is evident. Intersegmental connections, however, will result in antiphasic bursting behavior. These experiments indicate that the communication network is functioning properly.

VII. Conclusion

We developed an asynchronous architecture for modeling intersegmental neural communication that maintains neurobiological realism. This unique AER architecture includes a pipelined broadcast scheme that emulates a large number of intersegmental connections with distance-dependent delays. The intrasegmental units are half-center oscillators composed of silicon neurons, synaptic spread governs the interneuronal synaptic connections, and its implementation is simplified by using a relative addressing scheme, as opposed to a global bus architecture. The architecture is scalable, supports multichip communication, and operates independently of the type of silicon neuron (spiking or burst envelopes).

We are using this network to develop full intersegmental coordination systems that combine neural encodings with mechanical actuation [20]. In addition, we are developing more complex CPG circuits [14] and constructing hybrid neural systems that are composed of one of these more recently improved silicon neurons and a living heart interneuron from the heartbeat timing network of the medicinal leech [21].

Acknowledgment

The authors would like to thank R. Calabrese and A. Cohen for providing the biological expertise and inspiration for this system.

References

[1] J. Buchanan, "Identification of interneurons with contralateral, caudal axons in the lamprey spinal cord: Synaptic interaction and morphology," *J. Neurophys.*, vol. 47, pp. 961–975, 1982.

[2] C. A. Mead, *Analog VLSI and Neural Systems*. Reading, MA: Addison-Wesley, 1989.

[3] T. Williams, "Phase coupling and synaptic spread in chains of coupled neuronal oscillators," *Science*, vol. 258, pp. 662–665, 1992.

[4] K. A. Boahen, *Communicating Neuronal Ensembles Between Neuromorphic Chips*, ser. Neuromorphic Systems Engineering. Boston, MA: Kluwer, 1997, ch. 11, pp. 229–261.

[5] M. A. Mahowald, "VLSI analogs of neuronal visual processing: a synthesis of form and function," Ph.D. dissertation, California Inst. Technol., Pasadena, 1992.

[6] S. DeWeerth, G. Patel, M. Simoni, D. Schimmel, and R. Calabrese, "A VLSI architecture for modeling intersegmental coordination," in *Proc. 17th Conf. Advanced Research in VLSI*, A. Ishii and R. Brown, Eds., 1997, pp. 182–200.

[7] C. Morris and H. Lecar, "Voltage oscillations in the barnacle giant muscle fiber," *Biophys. J.*, vol. 35, no. 1, pp. 193–213, 1981.

[8] A. J. Martin, "Synthesis of asynchronous VLSI circuits," California Inst. Technol., Pasadena, 1991.

[9] ——, "Tomorrow's digital hardware will be asynchronous and verified," in *Proc. IFIP 12th World Computer Congr.*, vol. 1, J. van Leeuwen, R. Aiken, and V. Vogt, Eds., Madrid, Spain, Sep. 1992, pp. 684–695.

[10] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[11] F. Prosser, D. Winkel, and E. Brunvand, "A comparison of modular self-timed design styles," Comput. Sci. Dept., Indiana Univ., Indianapolis, IN, Tech. Rep. TR-420, 1994.

[12] M. A. Mahowald and R. Douglas, "A silicon neuron," *Nature*, vol. 354, no. 6354, pp. 515–518, Dec. 1991.

[13] G. Patel and S. DeWeerth, "Analogue VLSI Morris-Lecar neuron," *Electron. Lett.*, vol. 33, no. 12, pp. 997–998, 1997.

[14] M. F. Simoni, G. S. Cymbaluyk, M. E. Sorensen, R. L. Calabrese, and S. P. DeWeerth, "A multiconductance silicon neuron with biologically matched dynamics," *IEEE Trans. Biomed. Eng.*, vol. 51, no. 2, pp. 342–354, Feb. 2004.

[15] J. Lazzaro, J. Wawrzynek, M. Mahowald, M. Sivilotti, and D. Gillespie, "Silicon auditory processors as computer peripherals," *IEEE Trans. Neural Netw.*, vol. 4, no. 3, pp. 523–528, May 1993.

[16] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 47, no. 5, pp. 416–434, May 2000.

[17] C. Diorio, "Floating-gate MOSFETs," in *Analog VLSI: Circuits and Principles*, S.-C. Liu, J. Kramer, G. Indiveri, T. Delbrück, and R. Douglas, Eds. Cambridge, MA: MIT Press, 2002, ch. 4, pp. 93–120.

[18] R. R. Harrison, J. A. Bragg, P. Hasler, B. A. Minch, and S. P. DeWeerth, "A CMOS programmable analog memory-cell array using floating-gate circuits," *IEEE Trans. Circuits Syst. II, Analog Digit. Signal Process.*, vol. 48, no. 1, pp. 4–11, Jan. 2001.

[19] S. Grillner, P. Wallen, and L. Brodin, "Neuronal network generating locomotor behavior in lamprey: Circuitry, transmitters, membrane properties, and simulation," *Annu. Rev. Neurosci.*, vol. 14, pp. 169–199, 1991.

[20] M. F. Simoni, "Synthesis and analysis of a physical model of biological rhythmic motor control with sensorimotor feedback," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, 2002.

[21] M. Sorensen, S. DeWeerth, G. Cymbalyuk, and R. L. Calabrese, "Using a hybrid neural system to reveal regulation of neuronal network activity by an intrinsic current," *J. Neurosci.*, vol. 24, no. 23, pp. 5427–5438, Jun. 2004.

Girish N. Patel (S'98–M'99) received the B.S.E.E. degree from California Polytechnic, San Luis Obispo, in 1988, and the Ph.D. degree from the Georgia Institute of Technology, Atlanta, in 1999.

He has worked for Texas Instruments Incorporated and Microtune, and is currently with Alereon, Austin, TX.

Michael S. Reid (M'98) received the B.E.E. degree from Auburn University, Auburn, AL, in 1988, the M.B.A. degree from Carnegie Mellon University, Pittsburgh, PA, in 1994, and the M.S.E.E. degree from the Georgia Institute of Technology, Atlanta, in 2000. He is currently pursuing the Ph.D. degree in the School of Electrical and Computer Engineering at the Georgia Institute of Technology.

David E. Schimmel (S'82–M'90–SM'03) received the B.S.E.E. (with distinction) and Ph.D. degrees from Cornell University, Ithaca, NY, in 1984 and 1991, respectively.

He is currently Associate Professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology, Atlanta. He has been a Visiting Researcher at the University of Linkoping, Linkoping, Sweden, and a member of the summer faculty at NASA's Jet Propulsion Laboratory. He has also been a consultant to a number of corporations, including IBM Almaden Research Center and Intel. His research interests include parallel computer architecture and algorithms, VLSI design, asynchronous systems, and network hardware and software technologies.

Dr. Schimmel is a member of Tau Beta Pi and Eta Kappa Nu.

Stephen P. DeWeerth (S'85–M'90–SM'03) received the M.S. degree in computer science and the Ph.D. degree in computation and neural systems from the California Institute of Technology, Pasadena, in 1987 and 1991, respectively.

He is a Professor in the Wallace H. Coulter Department of Biomedical Engineering and in the School of Electrical and Computer Engineering at the Georgia Institute of Technology and at the Emory University School of Medicine, Atlanta, GA. His research focuses on the implementation of neuromorphic electronic and robotic systems, the development of neural interfacing technologies, and the study of the biological control of movement.